

Napredni razvoj programske podpore za web

**- predavanja -
2022./2023.**

1. TypeScript

Creative Commons



- slobodno smijete:
 - **dijeliti** — umnožavati, distribuirati i javnosti priopćavati djelo



- **prerađivati** djelo

- pod sljedećim uvjetima:

- **imenovanje:** morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
- **nekomercijalno:** ovo djelo ne smijete koristiti u komercijalne svrhe.
- **dijeli pod istim uvjetima:** ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.



U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.

Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.

Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.

Tekst licence preuzet je s <http://creativecommons.org/>

Što je TypeScript?

- TypeScript je nadgradnja JavaScripta u sintaksnom smislu (engl. *typed superset*) s ciljem izbjegavanja uobičajenih pogrešaka kakve se inače lako otkrivaju prevođenjem strogo tipiziranih programskih jezika
- TypeScript se prevodi/pretvara (engl. *transpile*) u JavaScript
 - Zadržava dinamička svojstva JS, a donosi tipsku sigurnost
 - Neki elementi TypeScripta postoje samo na razini prevodioca te se samo uklanjaju iz prevedenog koda, dok neki npr. generiraju dodatni JavaScript kod
- Raditi u TypeScriptu bez dobrog poznavanja JavaScripta nije rješenje problema, već samo veći problem!

Kako instalirati Typescript?

- Može se instalirati na više načina. U primjerima koji slijede koriste se *node.js* i *npm*, pa se instalacijska procedura za TypeScript svodi na

```
npm install -g typescript
```

uz provjeru instalirane verzije

```
tsc -v
```

- Instalacijske upute za neka druga razvojna okruženja dostupna su na

<https://www.typescriptlang.org/download>

Typescript i alternative

- Microsoft 2012., Anders Hejlsberg (C#, Pascal, Turbo Delphi)
- Mnoštvo pokušaja da se „poboljša” ili nadogradi JavaScript
 - <https://keyua.org/blog/top-alternatives-to-javascript-for-web-development/>
 - <https://www.sitepoint.com/10-languages-compile-javascript/>itd... ali praktično bez ozbiljne alternative
- Angular baziran nad njim, a moguće kombinirati i s Reactom i Vueom
- Deno kao poboljšana varijanta Node.js-a radi direktno s TypeScriptom

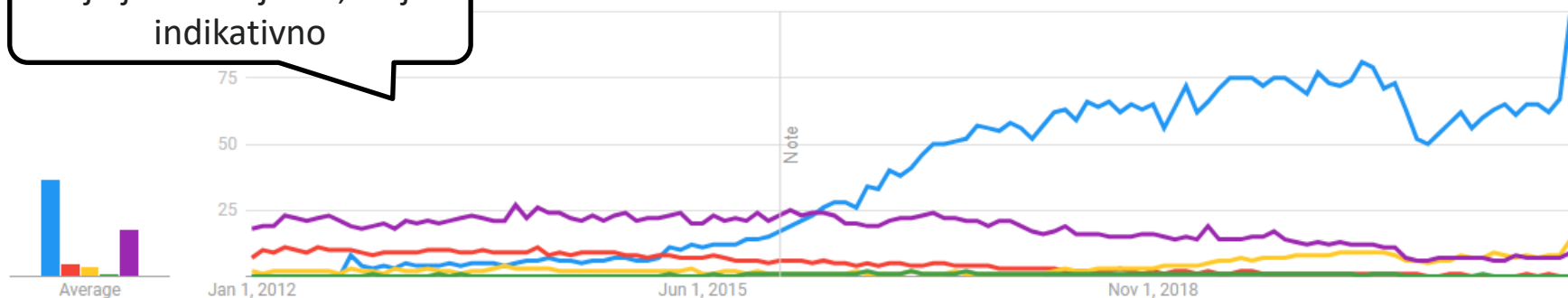
● TypeScript Programming language	● CoffeeScript Programming language	● Dart Programming language	● Elm Programming language	● Clojure Programming language
--------------------------------------	--	--------------------------------	-------------------------------	-----------------------------------

Worldwide ▼ 1/1/12 - 1/14/22 ▼ Programming ▼ Web Search ▼

Interest over time ?

<https://trends.google.com/trends/explore?cat=31&date=2012-01-01%202022-01-14&q=%2Fm%2F0n50hvx,%2Fm%2F0hjc5m0,%2Fm%2F0h52xr1,%2Fm%2F0ncc1sv,%2Fm%2F03yb8hb> ⬇ ⏪ ⏩ ⚡

Nije jedino mjerilo, ali je
indikativno



Motivacijski primjer (1)

00-intro-js/intro-incorrect.js

- Recepte učitavamo iz datoteke *recipes.json* tako da sastojke razdijelimo u polje stringova

```
const data = require("./recipes.json");
function load() {
  const recipes = data.map(function (item, index) {
    return {
      id: index + 1,
      ...item,
      recipeYield : Number(item.recipeYield),
      ingredients : item.ingredients ? item.ingredients.split("\n") : []
    }
  });
  return recipes;
}
```

```
[{
  "name": "Easter Leftover Sandwich",
  "ingredients": "12 whole Hard Boiled Eggs\n1/2 cup Mayonnaise\n ...",
  "url": "http://thepioneerwoman...", "image": "http://static.the...",
  "cookTime": "PT", "recipeYield": "8",
  "datePublished": "2013-04-01", "prepTime": "PT15M",
  "description": "Got leftover Easter eggs? ..."
},
{
  "name": "Pasta with Pesto Cream Sauce", ...
}
```

Motivacijski primjer (2)

00-intro-js/intro-incorrect.js

- Želimo pronaći sve recepte za 3-6 osoba koje sadrže sve navedene sastojke
- Pronađite pogreške u sljedećem kodu
 - Napomena: metoda *containsAllSubstrings* je vlastita metoda koja provjerava mogu li se svi stringovi iz polja zadanog kao drugi argument pronaći kao podnizovi u prvom polju

```
function findRecipes(yieldPredicate, ...ingredients) {  
  const recipes = load();  
  return recipes.filter(yieldPredicate)  
    .filter(r => containsAllSubstrings(r.ingredients, ingredients));  
}
```

```
const yieldPredicate = y => y >= 3 && y <= 6;  
const recipes = findRecipes(yieldPredicate, "Eggs", "Onion");  
recipes.forEach(r => {  
  console.log(`${r.id}. ${r.name} ${r.url} ${r.publishDate}`)  
});
```

```
node .\intro-incorrect.js
```

Motivacijski primjer zapisan u TypeScriptu (1)

- Kod za učitavanje recepata prebačen u zasebnu datoteku s ekstenzijom *ts*
 - Umjesto *require* koristi se *import* te je dodan *export* na kraju

```
import * as data from "./recipes.json"
function load() {
  const recipes = data.map(function (item, index) {
    return {
      id: index + 1,
      ...item,
      recipeYield : Number(item.recipeYield),
      ingredients : item.ingredients ? item.ingredients.split("\n") : []
    }
  });
  return recipes;
}
export default load;
```

00-intro-ts/loadrecipes.ts

Motivacijski primjer zapisan u TypeScriptu (2)

- Kod za učitavanje recepata prebačen u zasebnu datoteku s ekstenzijom *ts*

```
import loadrecipes from "./loadrecipes"
function findRecipes(yieldPredicate, ...ingredients) {
  const recipes = loadrecipes();
  return recipes.filter(yieldPredicate)
    .filter(r => containsAllSubstrings(r.ingredients,
                                      ingredients));
}
const yieldPredicate = y => y >= 3 && y <= 6;
const recipes = findRecipes(yieldPredicate, "Eggs", "Onion");
recipes.forEach(r => {
  console.log(`${r.id}. ${r.name} ${r.publishDate}`)
});
```

00-intro-ts/intro-start-from-incorrect.ts

VS Code: Cannot find name 'findRecipes'.
Did you mean 'findRecipes'?

Motivacijski primjer zapisan u TypeScriptu (3)

- Prethodni TypeScript kod možemo pretvoriti u JavaScript sljedećom naredbom (parametar `resolveJsonModule` omogućava korištenje modula iz *json* datoteka)

```
tsc intro-start-from-incorrect.ts --resolveJsonModule
```

što rezultira sljedećom porukom

```
00-intro-ts/...
```

```
intro-start-from-incorrect.ts:11:15 - error TS2552: Cannot find name  
'findRecipes'. Did you mean 'findRecipes'?
```

```
11 let recipes = findRecipes(yieldPredicate, "Eggs", "Onion");
```

~~~~~

```
intro-start-from-incorrect.ts:4:10
```

```
4 function findRecipes(yieldPredicate, ...ingredients) {
```

~~~~~

```
'findRecipes' is declared here.
```

```
Found 1 error.
```

- Unatoč tome, generira se js datoteka, što se može onemogućiti dodatnom opcijom `noEmitOnError`

```
tsc intro-start-from-incorrect.ts --resolveJsonModule --noEmitOnError
```

Motivacijski primjer zapisan u TypeScriptu (4)

- Ispravljenim pozivom metode prevodilac će uočiti novu pogrešku,

```
intro-start-from-incorrect.ts:12:50 - error TS2339: Property 'publishDate' does not exist on  
type '{ recipeYield: number; ingredients: string[]; name: string; url: string; image: string;  
cookTime: string; datePublished: string; prepTime: string; description: string; id: number; }'.  
12     console.log(`${r.id}. ${r.name} ${r.url} ${r.publishDate}`)
```

koju je također trivijalno za popraviti.

- Sve navedeno je posljedica toga što je TS automatski odredio koji tipovi se koriste u pojedinim izrazima i što oni sadrže
- TS ipak ne može sve, npr. neće otkriti pogrešku vezanu za korištenje predikata
 - Argument bi u ovoj varijanti trebao biti predikat za recept, a namjera pozivatelja je bila da bude predikat za broj osoba što zahtjeva modifikaciju

```
function findRecipes(yieldPredicate, ...ingredients) {  
    let recipes = load();  
    return recipes.filter(yieldPredicate)  
    ... //treba biti filter(r => yieldPredicate(r.recipeYield))  
}  
  
let yieldPredicate = y => y >= 3 && y <= 6;  
let recipes = findRecipes(yieldPredicate, "Eggs", "Onion");
```

Motivacijski primjer zapisan u TypeScriptu (5)

- Očita je namjera autora da *yieldPredicate* bude predikat za cijeli broj, pa se tako može i eksplicitno označiti u TS-u koristeći *type*
 - Lambda izrazom smo opisali kako mora izgledati tip (potpis) funkcije i pridijelili ime tom tipu
 - Ovaj način opisivanja funkcije se u literaturi naziva *function type expression*, a npr. funkcija napisana kao lambda *arrow function*
 - Dodatno, može se navesti od čega se sastoji ulazno polje, iako nije nužno

```
type numberPredicate = (x : number) => boolean;  
function findRecipes(yieldPredicate : numberPredicate, ...ingredients:string[]) {
```

- Nakon navedenog, prevodiocu je jasno da u filteru predikat ne može biti iskorišten na način na koji je napisano i da treba biti

```
recipes.filter(r => yieldPredicate(r.recipeYield))
```

- Ispravljena verzija nalazi se u `00-intro-ts/intro.ts`

Konfiguracijska datoteka za TS

- U prethodnim primjerima prilikom prevođenja navedeni su ime datoteke i parametri prevodioca
- Može se pojednostaviti korištenjem konfiguracijske datoteke *tsconfig.json* te se samo pokreće naredba *tsc*
 - Dodatno, postavljeno da generirane datoteke budu u mapi *dist* (isključena iz git repozitorija), a izvorni kod u *src*

```
{  
  "compilerOptions": {  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "noEmitOnError": true,  
    "resolveJsonModule": true  
  }  
}
```

00-intro-ts-proj/tsconfig.json

- U primjerima koji slijede sadržaj datoteke *tsconfig.json* će se mijenjati, a detaljnije o opcijama se može pronaći na
<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>
<https://www.typescriptlang.org/docs/handbook/compiler-options.html>

Tipovi podataka u JavaScriptu i TypeScriptu

- Podsjetnik: ugrađeni tipovi podataka u JS-u su
 - *number, bigint, string, boolean, null, undefined, object* i *symbol*
 - U JS-u tip varijable je određen pridruženom vrijednošću!
- Za razliku od JS-a (u koji će se prevesti), u TS-u je tip varijable eksplicitno naveden ili određen inicijalno pridruženom vrijednošću.
 - Tip varijable navodi se iza imena varijable

```
let x = 123;  
console.log(typeof(x)); // number  
x = "abc";  
console.log(typeof x); // string
```

```
let x : number = 123;
```

Prevodioc za typescript temeljem navedenog pridruživanja zaključuje da je x tipa number te se to ne treba eksplicitno navoditi.

Ovaj kod je valjan u JavaScriptu, ali ne i u Typescriptu, jer je varijabla x tipa *number* i (u Typescriptu) joj se ne može kasnije pridružiti vrijednost nekog drugog tipa.

Napomena: oznaka tipa postoji samo u .ts datoteci, odnosno u interpretaciji datoteke. U prevedenom kodu nema oznaka tipova podataka i ako nije uključen `noEmitOnError` onda ovo rezultira valjanim Javascript kodom.

Tip podataka *any*

- TS ne ograničava fleksibilnost JS-a, već pokušava spriječiti moguće pogreške
- Prethodni primjer je moguće napisati u TS-u, ako se tip podatka definira kao *any*

```
let x : any = 123;  
console.log(typeof x); // number  
x = "abc";  
console.log(typeof x); // string
```

- Koristi se za situacije kad je to zbilja potrebno, ali/jer uzrokuje gubitak tipske sigurnosti

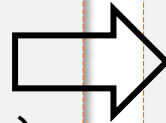
```
let x : any = 123;  
console.log(typeof x); // number  
console.log(x.length); // undefined  
x = "abc";  
console.log(typeof x); // string  
console.log(x.length); // 3
```

Implicitno definiran *any* (1)

- Ako nije eksplicitno definiran, TypeScript će pokušati zaključiti tip podatka temeljem pridružene vrijednosti

```
function f(v) {  
  let sum = 0;  
  v.forEach(a => sum += a * a);  
  return sum;  
}
```

01-implicit-any/src/index.ts



```
function f(v : any) : number {  
  let sum : number = 0;  
  v.forEach((a : any) => sum += a*a);  
  return sum;  
}
```

- Budući da prevodilac za TS ne zna što je *v*, onda ga smatra za *any*, pa je i *a* u lambda izrazu tipa *any*
- U opcijama prevodioca možemo uključiti opciju *declaration* kojom se generiraju datoteke s ekstenzijom *d.ts* u kojima možemo vidjeti kako je prevodilac interpretirao pojedine funkcije i varijable

```
{  
  01-implicit-any/tsconfig.json  
  "compilerOptions": {  
    "outDir": "./dist",  
    ...  
    "declaration": true,  
  }  
}
```

```
01-implicit-any/dist/index.ts.d  
declare function f(v: any): number;  
declare let x: number;
```


Implicitno definiran *any* (2)

- Posljedično, kad ulazni argument nije polje, prilikom izvršavanja dogodi se pogreška *TypeError: v.forEach is not a function*
- Prevodilac je temeljem operacija u funkciji zaključio da je *sum* broj, pa će smatrati *x* brojem
- Budući da se stringovi iz polja mogu automatski pretvoriti u brojeve, funkcija vraća istu (numeričku vrijednost) za polja iz primjera.
 - Ne zaboravite da u prevedenom kodu nema tipova podataka
 - Pokrenite program i promotrite rezultat, a zatim promotrite rezultat ako se izraz *sum+=a*a* izmijeni u *sum+=a*

```
function f(v) {  
  let sum = 0;  
  v.forEach(a => sum += a * a);  
  return sum;  
}
```

```
declare function f(v: any): number;  
declare let x: number;
```

01-implicit-any/src/index.ts

```
let x = f([1, 2, 3]);  
console.log(x);  
x = f(["1", "2", "3"]);  
console.log(x);  
x = f("abc");
```

- Implicitni *any* bi trebalo izbjegavati i može se onemogućiti uključivanjem opcije *noImplicitAny* prilikom prevođenja
- Slično vrijedi i za *noImplicitReturns*

Ako bi *v* bio *number[]*, tada bi prevodilac dojavio:
error TS2345: Argument of type 'string' is not assignable to parameter of type 'number[]'.

Postavke prevodioca – Provjera null vrijednosti

- Korisna opcija prevodioca je i provjera *null* vrijednosti za dijelove koda u kojima se *null* ne očekujem, kao u sljedećem primjeru

02-null-checks/*

```
{  
  "compilerOptions": {  
    "outDir": "./dist",  
    "rootDir": "./src",  
    "declaration": true,  
    "strictNullChecks": true,  
    "noImplicitAny": true  
  }  
}
```

```
function f(v : number[]) {  
  let sum = 0;  
  v.forEach(a => sum += a * a);  
  return Math.sqrt(sum);  
}  
  
let x = f([1, 2, 3]);  
let v : number[];  
x = f(v);
```

error TS2454: Variable 'v' is used before being assigned.

Unija kao tip podataka (1)

- Pretpostavimo da u JavaScriptu treba napisati metodu koja stvora novo polje kao permutaciju ulaznog polja.

```
function shuffle(data) {  
  let arr = [...data]; //novo polje  
  for (let i = arr.length - 1; i > 0; i--) {  
    const j = Math.floor(Math.random() * (i + 1));  
    const temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
  }  
  return arr;  
}
```

Moguće u
ES2015 ili novijoj
verziji

```
function shuffle(data) {  
  let arr = [...data]; //polje znakova  
  for (let i = arr.length - 1; i > 0; i--) {  
    const j = Math.floor(Math.random() * (i + 1));  
    const temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
  }  
  return arr.join('');  
}
```

Kôd metode koja bi
vratila permutirani string
ima tek neznatnu razliku

Unija kao tip podataka (2)

- Prethodni kod se može spojiti u jedan na sljedeći način

```
function shuffle(data) {  
  let arr = [...data];  
  for (let i = arr.length - 1; i > 0; i--) {  
    const j = Math.floor(Math.random() * (i + 1));  
    const temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
  }  
  return typeof (data) == "string" ? arr.join('') : arr;  
}
```

- Što ako želimo omogućiti da je ulazni argument ili string ili polje?

U tom slučaju ulazni argument je `string | any[]` što se naziva **unija**

- Može se pisati direktno unutar argumenta

03-union/src/index.ts

```
function shuffle(data : string | any[]) { ...
```

- ili definirati naziv za takav tip podataka i zatim ga koristiti

```
type stringOrArray = string | any[];  
function shuffle(data : stringOrArray) { ...
```

Definiranje povratne vrijednosti ovisno o argumentima

- Povratna vrijednost u prethodnom primjeru je `string | any[]`

```
function shuffle(data : string | any[]) : string | any[] {  
  ...  
}
```

- Ono što prevodilac ne zna, a autor funkcije zna, je činjenica da je povratna vrijednost *string* ako je ulaz *string*, odnosno polje, ako je ulazni argument polje.
- Nalik prototipu u C-u, možemo navesti sve dozvoljene kombinacije

03-union/src/index.ts

```
function shuffle(data : string) : string;  
function shuffle(data : any[]) : any[];  
function shuffle(data : string | any[]) : string | any[] {  
  ...  
}
```

Generički argumenti

```
function shuffle(data : string) : string;  
function shuffle(data : any[]) : any[];
```

- Problem s prethodnim rješenjem je u tome što ne čuva informaciju koji tip polja je korišten, već je povratni tip `any[]`
- TS omogućava generičke tipove podataka, pa se rješenje jednostavno modificira u

04-union-with-generics/src/index.ts

```
function shuffle(data : string) : string;  
function shuffle<T>(data : T[]) : T[];  
function shuffle<T>(data : string | T[]) : string | T[] {  
    ...  
    return typeof data == "string" ? arr.join('') : arr as T[];  
}
```

Umjesto `T[]` može se pisati i `Array<T>`

- Rad s *genericsima* je sličan kao i u Javi/C#-u, a omogućena je i parametrizacija JS kolekcija Map i Set
- Tipove u parametrizaciji moguće je ograničiti s *extends*
npr. `<T extends number | string>` ili
`<T extends {name: string, grade : number}>`

Tip koji ima navedena svojstva

<https://www.typescriptlang.org/docs/handbook/2/generics.html>

Ograničavanje mogućih vrijednosti varijable (1)

- Npr. želimo definirati tip podatka za automobil, pri čemu je jedna od vrijednosti oblik auta koji može poprimiti vrijednost iz ograničenog skupa vrijednosti.
- JS ne poznaje mogućnost enumeracije, ali u TS-u postoje enumeracije, koje se mogu izvesti na više načina
- Prvi pristup bi bio definiranje enumeracije nalik kako se to radi npr. u C-u

```
enum CarShape {  
    Sedan, Coupe, StationWagon, Hatchback, SUV, Other  
}
```

- Vrijednosti su cijeli brojevi počevši od 0. Slično kao u C-u moguće je eksplicitno pridružiti vrijednost pojedinom elementu, a onda sljedeći ima za jednu veću itd.
- Problem s ovim pristupom je način kako prevodilac generira JS kod što može imati utjecaj na performance

```
const shape : CarShape = CarShape.Sedan;
```

```
const shape = CarShape.Sedan;
```

```
var CarShape;  
(function (CarShape) {  
    CarShape[CarShape["Sedan"] = 0] = "Sedan";  
    CarShape[CarShape["Coupe"] = 1] = "Coupe";  
    CarShape[CarShape["StationWagon"] = 2] = "StationWagon";  
    CarShape[CarShape["Hatchback"] = 3] = "Hatchback";  
    CarShape[CarShape["SUV"] = 4] = "SUV";  
    CarShape[CarShape["Other"] = 5] = "Other";  
})(CarShape || (CarShape = {}));
```

Ograničavanje mogućih vrijednosti varijable (2)

- Enumeracije se mogu označiti kao konstantne

```
const enum CarShape {  
    Sedan, Coupe, StationWagon, Hatchback, SUV, Other  
}  
05-union-and-intersection/src/data.ts
```

- U ovom slučaju se prilikom prevođenja (osim ako nije uključena opcija *preserveConstEnums*), izrazi mijenjaju brojevima, a opis se stavljam u komentar

```
const shape : CarShape = CarShape.Coupe;  →  const shape = 1 /* Coupe */;
```

- Nijedna od ove dvije varijante ne sprječavaju korisnika da eksplicitno pridruži neki broj izvan raspona, pa se izvorni smisao enumeracija gubi.
- Ograničenje se može postići korištenjem stringova za vrijednosti,

```
enum CarShape {  
    Sedan = "S", Coupe = "C", StationWagon = "SW", Hatchback = "H", SUV = "SUV", Other = "O"  
}
```

te se u tom slučaju u TS kodu ne može pridružiti direktno string, već se mora koristiti enumeracija

- ponašanje s i bez *const* je ekvivalentno kao kod cijelih brojeva
- više o enumeracijama na <https://www.typescriptlang.org/docs/handbook/enums.html>

Ograničavanje mogućih vrijednosti varijable (3)

- Umjesto enumeracija mogu se razmotriti i drugačiji pristupi, kao npr. unije

```
type CarShape = "Sedan" | "Coupe" | "SW" | "H" | "SUV" | "0";
```

```
const shape : CarShape = CarShape.Coupe; → const shape = "Coupe";
```

ili konstantni objekti

```
type CarShape = typeof CarShapes[keyof typeof CarShapes];  
const CarShapes = { Sedan : 0, Coupe : 1, StationWagon : 2, Hatchback : 3,  
  SUV : 4, Other : 5 }
```

```
const shape : CarShape = CarShapes.Coupe; → const shape = CarShapes.Coupe
```

- Napomena:
 - Operator `keyof` vraća uniju naziva svojstava nekog tipa čime definira novi tip. `CarShapes` je objekt čiji nas tip zanima, pa je `keyof typeof CarShapes` tip definiran kao `"Sedan" | "Coupe" | "StationWagon" | "Hatchback" | "SUV" | "Other"`
 - Nakon toga `typeof nekiobjekt[tip]` vraća uniju tipova podataka svojstava čiji su nazivi činili tip naveden u uglatim zagradama. Posljedično, `CarShape` je `number`
 - Usput, `tip[keyof tip]` je konstrukcija koja bi vratila uniju svih tipova koje neki tip sadrži. Detaljnije na <https://www.typescriptlang.org/docs/handbook/2/indexed-access-types.html>

Spajanje tipova

- Osobim unije tipova, može se izvesti i spajanje tipova

```
type Car = {  
  registration: string  
  brand: string  
  model: string  
  shape: CarShape  
}
```

```
type Owner = {  
  name: string  
  registration: string  
  sex: 'M' | 'F'  
}
```

05-union-and-intersection/src/*.ts

```
type Registration = Car | Owner; // union  
type FullData = Car & Owner; // intersection
```

- Napomena: nazivi ovih konstrukcija mogu biti zbunjujući, naročito naziv *presjek* koji predstavlja spajanje čime nastaje novi tip koji sadrži sva svojstva iz oba tipa
- Ako prilikom spajanja u oba tipa postoji svojstvo istog imena, onda se na tip tog svojstva primjenjuje operator &
 - Za primitivne tipove to nema smisla, pa je tako `number & string` isto što i `never`
 - Za objekte je takvo spajanje smislenije npr. `{price: number} & {name: string}` će tvoriti tip koji ima svojstva *price* i *name*, što je praktičan način za stvaranje novih tipova proširenjem postojećih, npr.

```
type CarWithPrice = Car & {price: number}
```

Provjera tipa (type guard) (1)

- Kod primitivnih tipova, provjeru tipa možemo obaviti s `typeof v`
- Provjera je li nešto polje radi se s `Array.isArray(v)`, gdje je `v` neka varijabla.
- U slučaju kad imamo uniju objekata provjera se svodi na ispitivanje sadrži li trenutni objekt određeno svojstvo operatorom `in`
- Napravi li se provjera temeljem svojstva koje korektno diskriminira moguće tipove iz unije, prevodilac će točno znati s kojim tipom radi u nastavku
 - U literaturi se ovaj princip naziva sužavanje (*narrowing*), a izraz provjere *type guard*

```
function registrationSet(...registrations : Registration[]) : Set<Registration> {  
  const set = new Set<Registration>();  
  registrations.forEach(r => {  
    set.add(r);  
    //what we have here?  
    if ("brand" in r)  
      console.log(`${r.registration} from Car ${r.model}`);  
    else  
      console.log(`${r.registration} from Owner ${r.name}`);  
  });  
  return set;  
}
```

05-union-and-intersection/src/*.ts

Provjera tipa (type guard) (2)

- Provjera se može obaviti i u funkciji koja se naziva *type predicate* te završava s : *parametar is neki tip*

```
function isCar(r : Registration) : r is Car {  
    return "brand" in r; //ostale provjere izostavljene zbog jednostavnosti  
}
```

05-union-and-intersection/src/*.ts

```
function registrationSet(...registrations : Registration[]) : Set<Registration> {  
    const set = new Set<Registration>();  
    registrations.forEach(r => {  
        set.add(r);  
        //what we have here?  
        if (isCar(r))  
            console.log(`${r.registration} from Car ${r.model}`);  
        else  
            console.log(`${r.registration} from Owner ${r.name}`);  
    })  
    ...  
}
```

n-torke

- TS dozvoljava definiranje n-torki (različitih tipova)
- Npr. vektor u 3D prostoru, možemo definirati kao trojku brojeva
`type vector = [number, number, number];` 06-tuples/src/vectors.ts
- Posljedično će prevodilac znati da vektor ima elemente samo na pozicijama 0, 1 i 2 te neće dopustiti definiranje vektora s manje ili više od 3 elementa

```
let a : vector = [2, 3, 5];  
let b : vector = [-1, 4, 6];  
//let impossible : vector = [-1, 4, 6, 2];  
//let error = a[3];
```

- Tipovi mogu biti različiti, a oni navođeni na kraju n-torke mogu biti i opcionalni, ali će prevodilac na to upozoriti

```
type n4 = [string, number | string, boolean?, number?];  
const a : n4 = ["A", 3, true];  
const b : n4 = ["B", 2];  
const val : boolean = a[2]; //const val : boolean = a[2]!;
```

error TS2322: Type 'boolean | undefined' is not assignable to type 'boolean'.

- Napomena: n-torka se implementira kao polje, pa je moguće pozvati *pop* i *push* što narušava ideju n-torki, ali sugestija za onemogućavanje nije usvojena
 - <https://github.com/microsoft/TypeScript/issues/6325>

Sučelja

- Osim s n-torkom, vektor u 3D prostoru se može definirati koristeći *type*
`type vector = { x: number, y: number, z: number }`
ali i koristeći sučelja
`interface Vector { x: number; y: number; z: number; }`
- Za razliku od klasičnih OOP jezika u TS-u se provjera tipa svodi na provjeru postojanja potrebnih svojstava
 - structural subtyping, duck typing, shape matching, soundness...
 - <https://www.typescriptlang.org/docs/handbook/type-compatibility.html>

```
interface Vector { x: number; y: number; z: number; }
type vector = { x: number; y: number; z: number; }
function t_print(v: vector) { console.log(v); }
function i_print(v: Vector) { console.log(v); }
function a_print(v: {x: number, y: number, z: number}) { console.log(v); }
let iv : Vector = {x : 3, y : 5, z : 2};
let tv : vector = {x : 1, y : 2, z : 3};
t_print(iv); i_print(tv); a_print(iv);
let temp = iv; iv = tv; tv = temp;
let w = {x : 9, y : 8, z : 7, w : 6};
iv = w;
i_print(iv);
```

07-types-and-interface/src/index.ts

Sučelja i aliasi

- U izoliranom primjeru, korištenje *type* i interface daje isti efekt.
- *type* predstavlja alias za neki tip podatka, dok je interface još jedan način za definiranje tipa objekta.
- Sučelje može naslijediti drugu sučelje s *extends*, aliasi to rade s &
- Klase mogu implementirati i sučelja i aliase
 - Naravno, ako je alias zapravo alias na neki objekt, a ne alias primitivnog tipa
- Ključne razlike:
 - Sučelje se može definirati više puta, što ima efekt dodavanja novih svojstava
 - Kod aliasa to nije moguće
 - Neke manje razlike u prikazu pogreške prilikom prevođenja
 - <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#differences-between-type-aliases-and-interfaces>

Klase u TypeScriptu

- JS (od ES2015), pa tako i TS podržavaju klase, ali na ponešto drugačiji način nego u klasičnim objektno orijentiranim jezicima
- objekti u memoriji ne nose informaciju iz koje klase ili sučelja su potekli (*not reified*) te će `typeof (new Klasa())` uvijek biti `object`
- Članovi klase su *public* osim ako se drugačije ne navedene (*private* ili *protected*)
 - modifikatori vidljivosti vrijede samo prilikom prevođenja! ...
 - osim ako se ne koristi novi standard s prefiksom `#` za privatne varijable
 - ... a imaju nuspojavu da utječu na provjeru tipova
 - Podsjetnik: provjera tipa svodi se na provjeru postojanja potrebnih svojstava, odnosno funkcija, a *private* „mijenja” oblik
- Sugerira se definiranje varijabli javnima u situacijama kad bi *getter* i *setter* bili trivijalni
- Pristup članovima unutar klase uvijek mora biti s *this.ime*, jer samo *ime* bi moglo biti referenca na varijablu definirano negdje drugdje

Konstruktor(i)

- JS dozvoljava postojanje samo jednog konstruktora
- U TS-u se može definirati više konstruktora, ali na razini dozvoljenih varijanti poziva jedinstvene implementacije!

```
class Vector {  
    x:number; y:number; z:number;  
    constructor(x: number, y:number, z:number)  
    constructor(vector : {x : number, y : number, z : number});  
    constructor(vector: [number, number, number]);  
    constructor(data : number | {x : number, y : number, z : number}  
                | [number, number, number],  
                ...yz : number[]) {  
        if (typeof(data) === "number") {  
            this.x = data; this.y = yz[0]; this.z = yz[1];  
        }  
        else if (Array.isArray(data))  
            [this.x, this.y, this.z] = data;  
        else {  
            this.x = data.x; this.y = data.y; this.z = data.z;  
        }  
    }  
}
```

08-classes/src/vector.ts

Inicijalizacija varijabli

- Postavkama prevodioca može se uključiti opcija koja provjerava jesu li varijable inicijalizirane u konstruktoru ili prilikom deklaracije.
- Neovisno o tome, varijable koje se nakon postavljanja (pri deklaraciji ili u konstruktoru) više ne mijenjaju mogu se označiti s *readonly*, pa će prevodilac prijaviti pogrešku ako im se naknadno pokuša promijeniti vrijednost

```
{  
  "compilerOptions": {  
    "target": "ES2015",  
    "module": "CommonJS",  
    "outDir": "./dist",  
    "rootDir": "./src",  
    ...  
    "strictPropertyInitialization" : true,  
    "noEmitOnError": true  
  }  
}
```

Svojstva su podržana tek od verzije ES2015. *target* za sobom onda povlači i *module*

08-classes/tsconfig.json

Deklaracija članskih varijabli temeljem argumenata konstruktora

- U slučaju da je klasa imala samo jedan konstruktor koji bi se sveo samo na inicijalizaciju članskih varijabli, tada se može izbjeći deklariranje varijabli i pisanje koda za pridruživanje.

```
class ImmutableVector {  
    readonly x:number; readonly y:number; readonly z:number;  
    constructor(x: number, y:number, z:number) {  
        this.x = x; this.y = y; this.z = z;  
    }  
}
```

- Sve što je potrebno je dodati modifikatore ispred argumenata (*private*, *protected* ili *public* te opcionalno *readonly*)

```
class ImmutableVector {  
    constructor(public readonly x: number, public readonly y:number,  
                public readonly z:number) {  
    }  
}
```

Svojstva i metode

- Svojstvo izgleda kao metoda koja ispred ima *get* ili *set*, a pri pozivu se zagrada ispušta
 - Ako postoji *setter*, mora imati istu vidljivost kao i *getter*, ali ne moraju imati iste tipove (!?). Ako nije naveden tip za *setter*, koristi se onaj od *gettera*.
 - Svojstva podržana od ES2015

```
class Vector implements IHasNorm, IComparable<Vector> {  
  x:number; y:number; z:number;  
  ...  
  get norm() : number {  
    return Math.sqrt(this.x ** 2 + this.y ** 2 + this.z ** 2);  
  }  
  
  scalar(other : Vector) : number {  
    return this.x * other.x + this.y * other.y + this.z * other.z  
  }  
  
  cross(other : Vector) : Vector {  
    ...  
  }  
}
```

08-classes/src/*.ts

```
let a = new Vector(2, 3, 5);  
let b = new Vector([-1, 4, 6]);  
let c = a.cross(b);  
console.log(`c=${c}`);  
console.log(`|c|=${c.norm}`);
```

Sažetak vezan za provjere i pretvorbe tipova

- `typeof` vraća jedan od sljedećih stringova: *string, number, bigint, boolean, symbol, undefined, object, function*
- `instanceof` se može koristiti samo za klase i istina je samo ako je objekt stvoren s `new`
- Sužavanje (engl. *narrowing*) kod objekata se vrši operatorom `in` i provjerom sadrži li objekt navedeno svojstvo
- Tvrdnja da je neki općeniti tip (*any*) nešto konkretno, može se obaviti s `as` ili operatorom ukalupljivanja
 - npr. `x as Vector` ili `<Vector> x`
 - Navedeno ima smisla samo prilikom pisanja koda i prevođenja
 - Slična tvrdnja vrijedi i za operator `!` kojim prevodiocu tvrdimo da objekt nije *null*
- Izraz unutar *if*-a ne mora biti *boolean*, već se evaluira po pravilu da su laži *0, NaN, prazni string, null* i *undefined*

Transformacije tipova u nove tipove podataka

- Stvaranje novih tipova iz postojećih ne mora se svoditi samo na proširivanje
- TS nudi nekoliko ugrađenih generičkih tipova kojima nastaje novi tip iz postojećeg uklanjanjem nekih svojstava, odabirom samo određenih, promjenom imena...
- Npr. želimo stvoriti novi tip iz tipa *Recipe* (a on je određen datotekom s receptima), tako da odaberemo samo *name* i *description* uz dodatno opcionalno svojstvo
 - Ovdje su navedena svojstva bila direktno napisana, ali u nekom drugom slučaju mogli smo ih dobiti s *keyof* nekog poznatog objekta
- Suprotan od *Pick* je *Omit* kojim uzimamo sva svojstva osim onih koje eksplicitno navedenih. *Partial*, *Required* i *Readonly* mijenjaju modifikatore svojstava. Za ostale pogledati na <https://www.typescriptlang.org/docs/handbook/utility-types.html>

```
import {load as loadrecipes} from "./loadrecipes"
type Recipe = typeof recipes[0];
type RecipeInfo = Pick<Recipe, "name" | "description"> & {whereToEat? : string};
let recipes = loadrecipes();
```

09-various-type-creation/src/index.ts

Dinamičko dodavanje svojstava

- Tipična funkcionalnost u JS-u je dodavanje novog svojstva, ali u TS-u bi sljedeći kod bio neispravan, jer svojstvo Zagreb nije dio tipa za navedenu varijabla

```
let citiesAndMeals = {};  
citiesAndMeals.Zagreb = nešto... //citiesAndMeals["Zagreb"] = ...
```

- U takvim slučajevima koristi se *index signature* kojim se definira da se tom tipu podatka mogu dinamički dodavati svojstva sve dok zadovoljavaju tipove određene potpisom za indeks
 - U primjeru to znači da se u uglatim zagrada prilikom dodavanja novog svojstva mogu naći stringovi (ali i brojevi, zbog automatske pretvorbe), a pridružene vrijednosti mora biti tipa *RecipeInfo*
 - Moguće je koristiti i unije i presjeke, ali ključ može biti samo kombinacija tipova *string*, *number* i *boolean*
 - Sučelje može definirati i dodatna svojstva, ali ona moraju biti u skladu s indeksom**

```
interface CitiesTopMeal {  
  [key:string] : RecipeInfo; //index signature  
};  
let citiesAndMeals : CitiesTopMeal = {};  
citiesAndMeals.Zagreb = nešto... //OK
```

09-various-type-creation/src/index.ts

Što dalje?

- Typescript je previše opsežan da bi se sve mogućnosti opisale u jednom predavanju
 - *MappedTypes, Iterators, ConditionalTypes*, uključivanja JavaScript koda u proces prevođenja, integracije s Reactom, Vueom, ...
- Za one koji žele znati više
 - <https://www.typescriptlang.org/docs/handbook/intro.html>
 - <https://www.typescriptlang.org/cheatsheets>
 - Adam Freeman: *Essential TypeScript 4. From Beginner to Pro*, Apress 2021
<https://link.springer.com/book/10.1007/978-1-4842-7011-0>